

PPSS PLANIFICACIÓN Y PRUEBAS DE SISTEMAS SOFTWARE

Curso 2020-21

Sesión S01: Pruebas: proceso de diseño: caja blanca



Pruebas y proceso de pruebas.

Diseño de casos de prueba: structural testing

- Objetivo: obtener una tabla de casos de prueba a partir del conjunto de comportamientos programados
- El conjunto obtenido debe detectar el máximo número posible de defectos en el código, con el mínimo número posible de "filas"

Control flow testing: Método del camino básico

- Paso 1: Análisis de código: Construcción del CFG
- Paso 2: Selección de caminos: Cálculo de CC y caminos independientes
- Paso 3. Obtención del conjunto de casos de prueba

Ejemplos y ejercicios

Vamos al laboratorio...

DEFINICIÓN DEL PROCESO DE PRUEBAS

“ “

" Testing es el proceso de ejecutar un programa con la intención de **encontrar errores**. Si nuestro objetivo es mostrar la ausencia de errores, descubriremos menos de los que esperamos. Si nuestro objetivo es mostrar la presencia de errores, descubriremos un gran número de ellos."

Glenford J. Myers (1979)

” ”

” ”

Las **pruebas** son un conjunto de actividades conducentes a **conseguir** alguno de estos **objetivos**:

- * **Encontrar defectos**
- * **Evaluuar el nivel de calidad del software**
- * **Obtener información para la toma de decisiones**
- * **Prevenir defectos**

(ISQTB Foundation LevelSyllabus -2011)

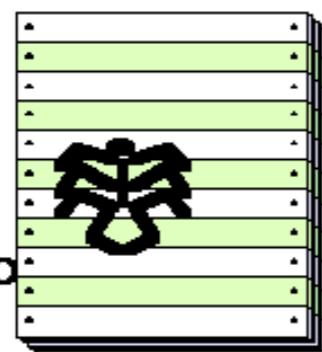
” ”

HAY MUCHOS TIPOS DE "ERRORES"!!



?

can lead to



human error
mistake

can lead to



fault
bug, defect

failure
error

Las pruebas muestran la **PRESENCIA de defectos** (no pueden demostrar la ausencia de los mismos. Si no se encuentra un defecto, no significa que no los haya)



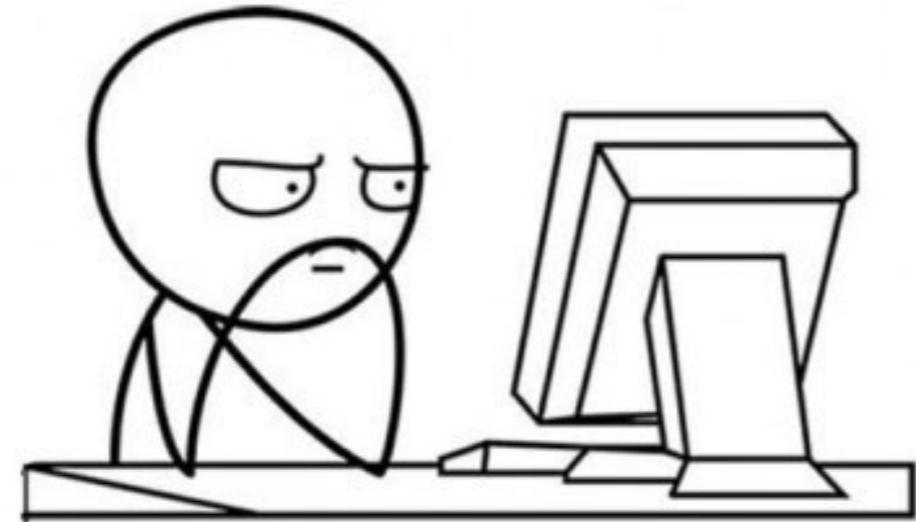
DEFINICIÓN DEL PROCESO DE PRUEBAS

P

P

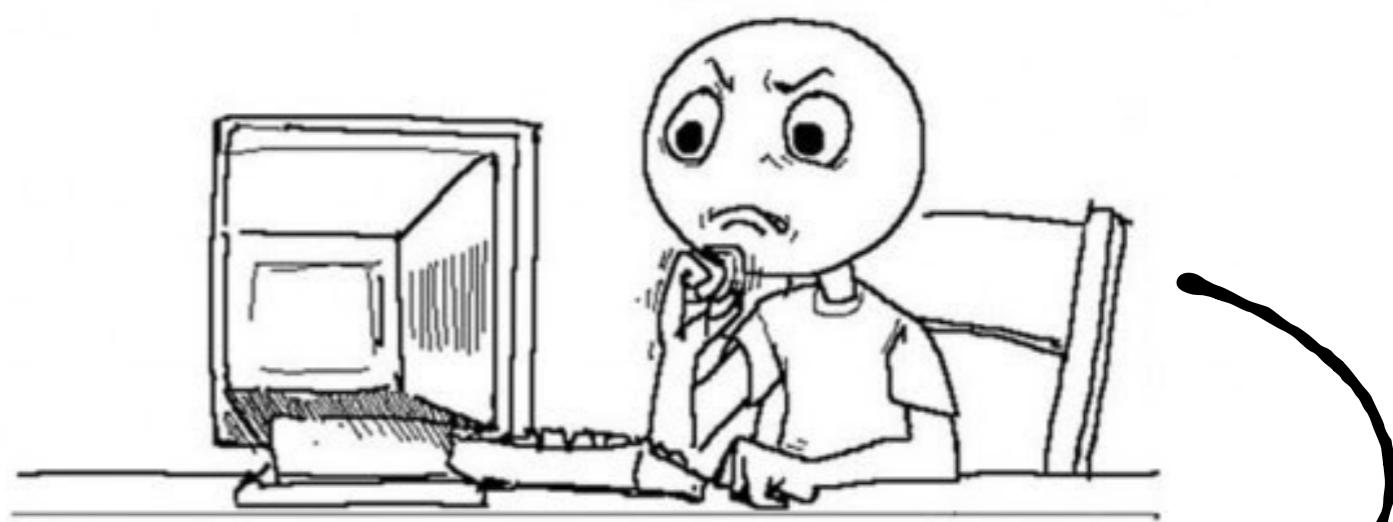
DEVELOPER

How can I make it?



TESTER

How can I break it?



A LOS USUARIOS NO LES GUSTAN LOS ERRORES!!



Uno de los **objetivos de éxito** del **proyecto** es que el **software satisfaga las expectativas del cliente**

Si las **expectativas del cliente no se satisfacen**, éste **se sentirá justificadamente agraviado**

Una prueba tiene **ÉXITO** cuando revela la **PRESENCIA de defectos**.

Además, no es suficiente el encontrar defectos, el **programa debe satisfacer las necesidades y expectativas del cliente**)



ACTIVIDADES DEL PROCESO DE PRUEBAS



SEGÚN ISQTB FOUNDATION LEVEL SYLLABUS (2011)

1 PLANIFICACIÓN y control de las pruebas

Definimos los objetivos de las pruebas, y en todo momento tenemos que asegurarnos de que cumplimos con esos objetivos (p.ej. queremos realizar pruebas sobre el 95% de código)

2 DISEÑO de las pruebas

Es el proceso más importante, si queremos efectivamente cumplir los objetivos marcados. Básicamente consiste en decidir con qué datos de entrada concretos vamos a probar el código, de forma que seamos capaces de detectar el máximo número de errores posibles, en el menor tiempo posible

3 IMPLEMENTACIÓN y ejecución de las pruebas

Creamos código, para probar nuestro código!!



...No, no nos hemos vuelto locos. La idea es que podemos ejecutar las pruebas pulsando un botón, en lugar de hacerlo de forma "manual". Lógicamente, hay que prestarle mucha atención al código de pruebas para que efectivamente nos ayude a conseguir nuestro objetivo

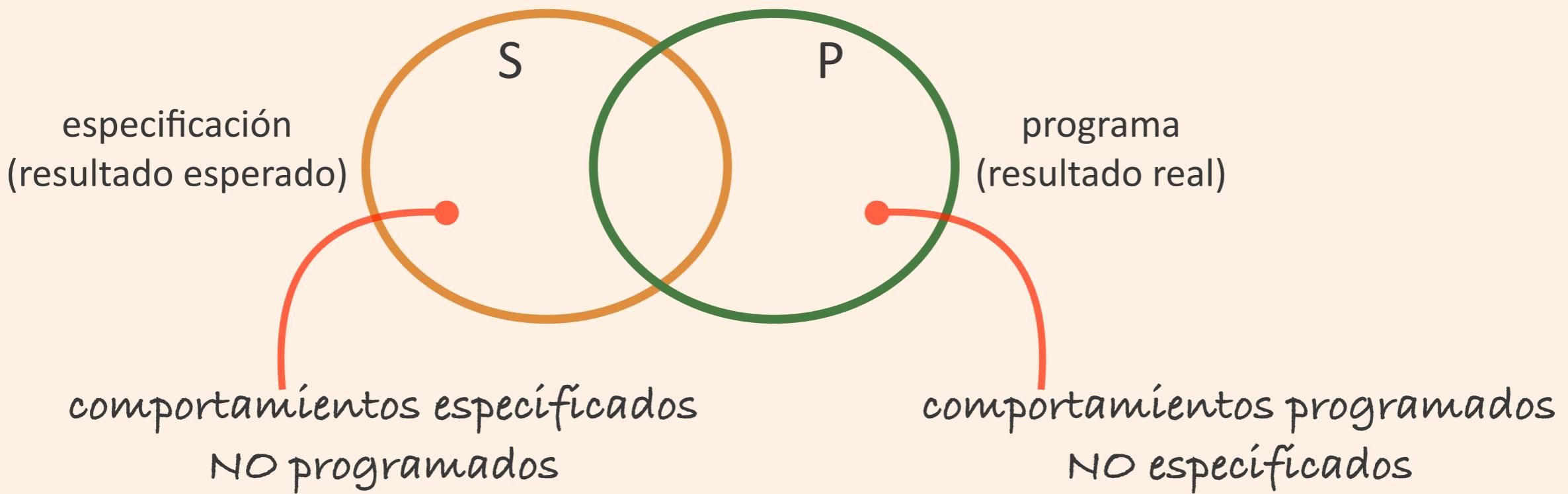
4 EVALUACIÓN del proceso de pruebas y emitir un informe

Aplicamos las métricas adecuadas para comprobar si hemos alcanzado los objetivos de pruebas planificados

PRUEBAS Y COMPORTAMIENTO

S y P deberían ser idénticos!!!

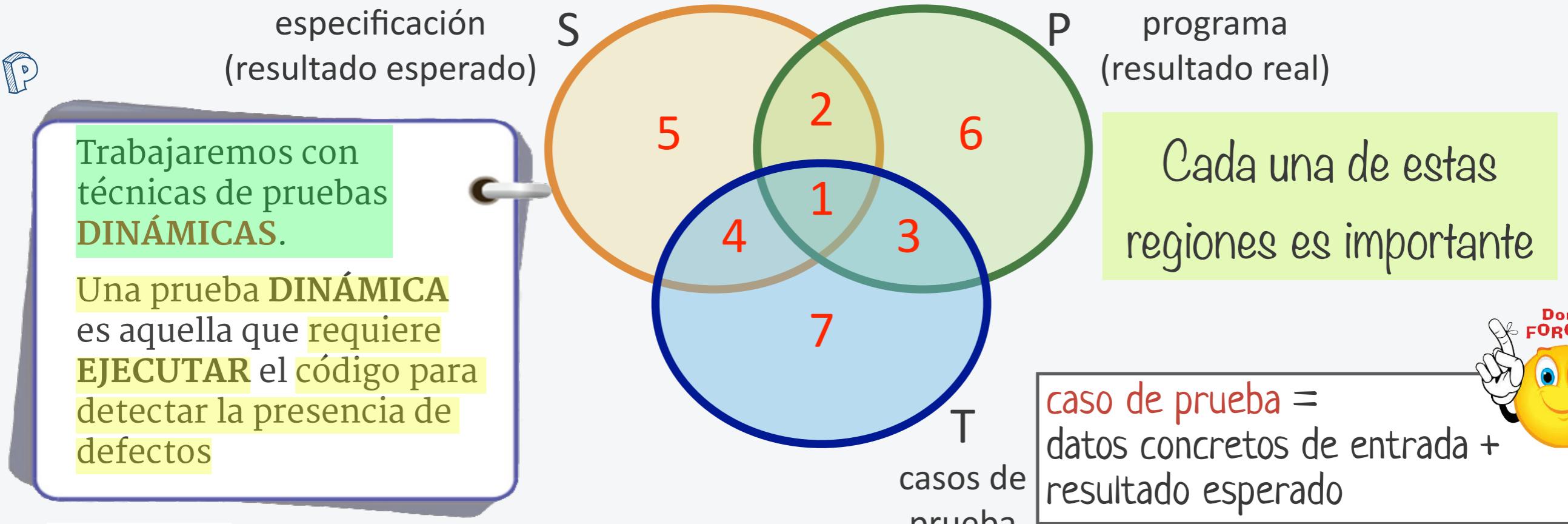
- Las pruebas conciernen fundamentalmente al comportamiento del elemento a probar: ¿qué debería hacer aquello que estoy probando?
- Supongamos un universo de comportamientos de programa. Dado un programa y su especificación, consideraremos:
 - el conjunto S de comportamientos especificados para dicho programa
 - el conjunto P de comportamientos programados



Estos son los problemas con los que se enfrenta un tester!!!

COMPORTAMIENTOS ESPECIFICADOS, PROGRAMADOS, Y PROBADOS

- Incluyamos el conjunto T de comportamientos probados a la figura anterior:



Indica qué representan las regiones:

- 1** 2+5 **2** 1+4 **3** 3+7 **4** 2+6 **5** 1+3 **6** 4+7

- ¿Qué puede hacer un tester para conseguir que la región 1 sea lo más grande posible?

→ Identificar un conjunto de **casos de prueba** utilizando algún método de DISEÑO de pruebas (el **objetivo** es encontrar el **máximo número** posible de **defectos** con el **mínimo número** de **casos de prueba**)

DISEÑO DE CASOS DE PRUEBA

S Selección de un subconjunto **mínimo** de entradas para evidenciar el **máximo** número de errores posibles

P El resultado del proceso de diseño es una **Tabla de casos de prueba**. Cada fila contiene los datos de entrada y el resultado esperado de un comportamiento del programa

Tábla de casos de prueba

ID	d1	d2	...	Expected Output
C1	?	?	?	?
C2				
C3				
...				
CN?				



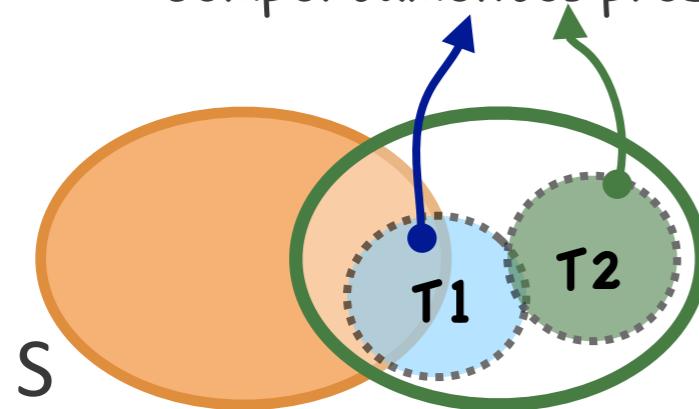
- ¿COMO RELLENAMOS LAS FILAS DE LA TABLA?
- ¿CUÁNTAS FILAS ES NECESARIO AÑADIR?

Utilizando un **MÉTODO** de **DISEÑO** de casos de prueba!!!!!!

Cada **FILA** de la tabla es un **CASO DE PRUEBA** = **datos entrada concretos + resultado esperado**

STRUCTURAL TESTING = DISEÑO DE PRUEBAS DE CAJA BLANCA

Comportamientos probados



Comportamientos implementados

Podemos detectar comportamientos no especificados
Nunca podremos detectar comportamientos no implementados

CN

- Consiste en determinar los **valores de entrada** de los casos de prueba a partir de la **IMPLEMENTACIÓN**.
- El **resultado esperado** se obtiene SIEMPRE de la especificación
- Los **comportamientos probados** podrán estar o no especificados
- Es esencial conocer conceptos de teoría de grafos para entender bien esta aproximación



CUALQUIER MÉTODO BASADO EN EL CÓDIGO SIGUE ESTOS PRINCIPIOS!!!



Los métodos de diseño de casos de prueba basados en el CÓDIGO:

1. Analizan el código y obtienen una representación en forma de GRAFO
2. Seleccionan un conjunto de caminos en el grafo según algún criterio
3. Obtienen un conjunto de casos de prueba que ejercitan dichos caminos

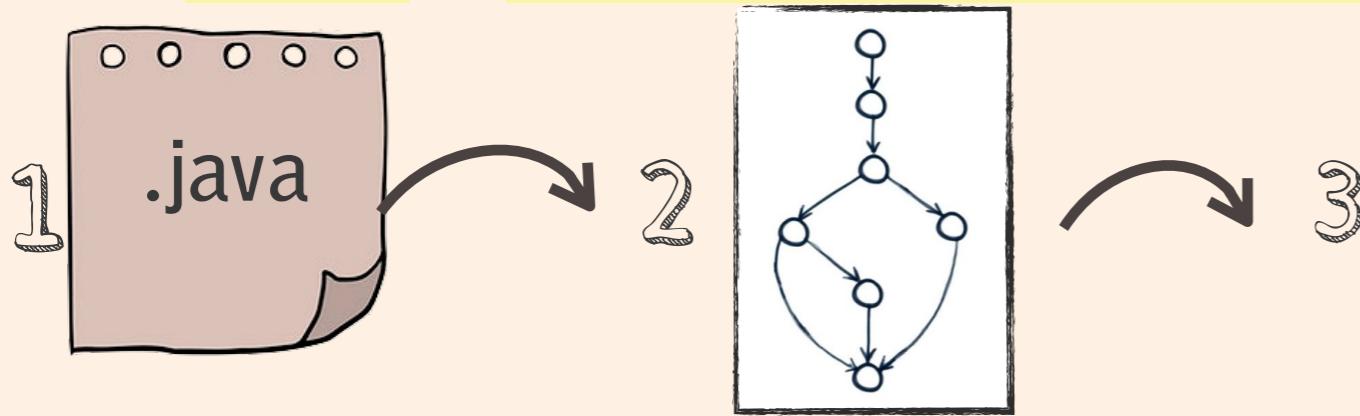


Tabla de casos de prueba

Test case	a	b	c	outcome

OBSERVACIONES SOBRE LOS MÉTODOS ESTRUCTURALES

- Dependiendo del método concreto utilizado, obtendremos conjuntos DIFERENTES de casos de prueba. Pero el conjunto obtenido será EFECTIVO y EFICIENTE!!!!
- Las técnicas o métodos estructurales son costosos de aplicar, por lo que suelen usarse sólo a nivel de UNIDADES de programa
- Los métodos estructurales NO pueden DETECTAR TODOS los defectos en el programa (defecto = fault = bug). Aunque seleccionemos todas las posibles entradas, no podremos detectar todos los defectos si "faltan caminos" en el programa.
De forma intuitiva, diremos que falta un camino en el programa si no existe el código para manejar una determinada condición de entrada.
Por ejemplo: si la implementación no prevé que un divisor pueda tener un valor cero, entonces no se incluirá el código necesario para manejar esta situación

P ANÁLISIS DE CÓDIGO BASADO EN EL FLUJO DE CONTROL

- Los dos tipos de sentencias básicas en un programa son.

- Sentencias de **asignación**

Por defecto se ejecutan de forma secuencial

- Sentencias **condicionales**

Alteran el flujo de control secuencial en un programa

- Las llamadas a "funciones" son un mecanismo para proporcionar abstracción en el diseño de un programa

- Una llamada a una "función" cede el control a dicha función

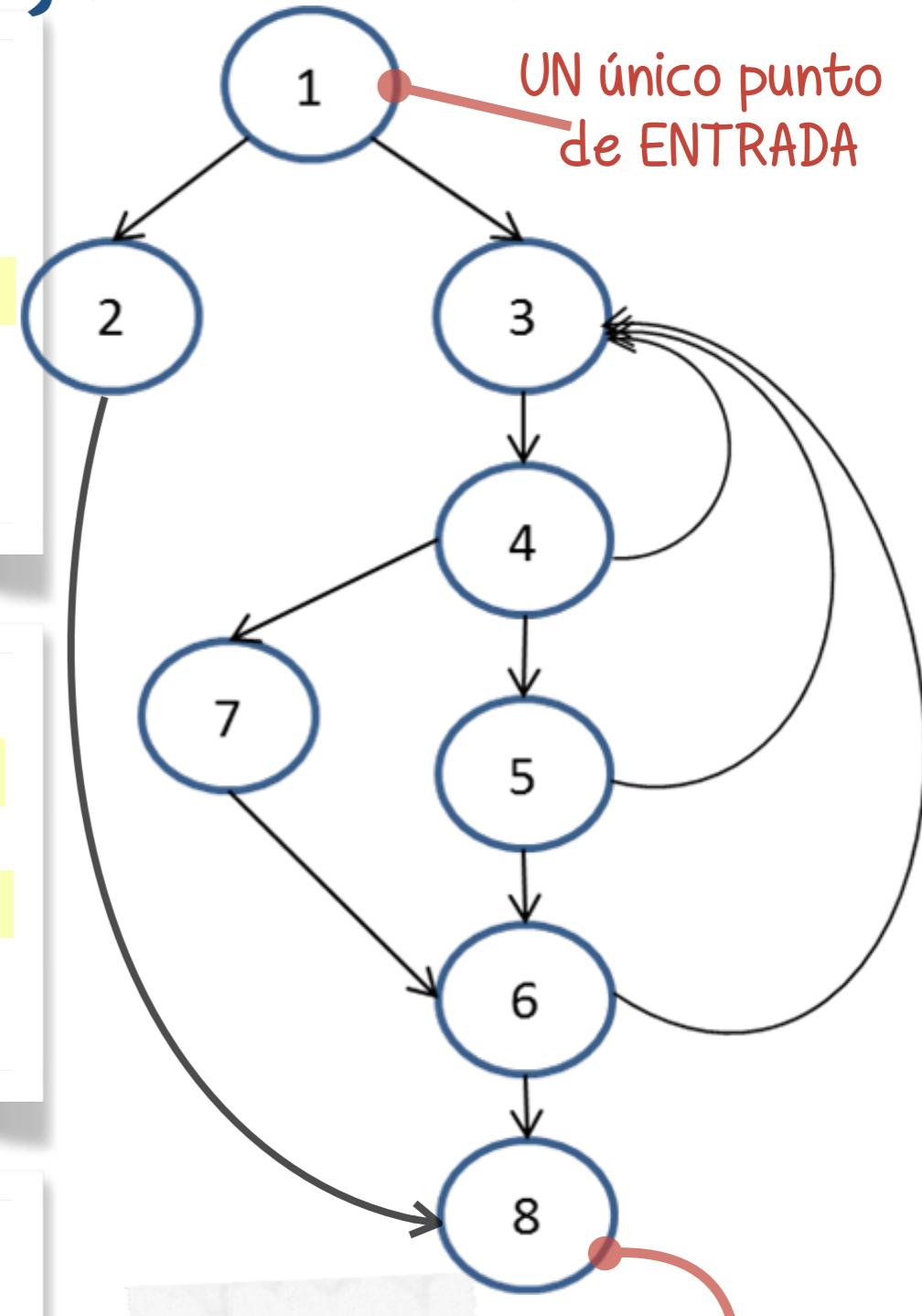
- Dentro de la unidad a probar, la invocación de una "función" es una sentencia secuencial

Cada camino en el grafo se corresponde con un **COMPORTAMIENTO!!!**

La **EJECUCIÓN** de una secuencia de instrucciones desde el punto de entrada al de salida de una unidad de programa se denomina **CAMINO** (path)

En una unidad de programa puede haber un número potencialmente grande de caminos, incluso infinito.

Un conjunto de valores específicos de entrada provoca que se ejecute un camino específico en el programa



UNIDAD de programa = método Java

UN único punto de ENTRADA

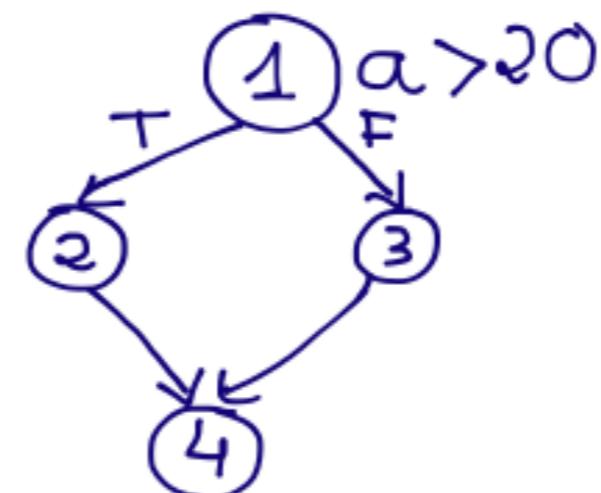
UN único punto de SALIDA

GRAFO DE FLUJO DE CONTROL (CFG)

P
S
P
S
1 nodo representa cero o más sentencias secuenciales + cero o UNA!! condición

- Un CFG es una **representación gráfica** de una **unidad** de programa. Usaremos un **GRAFO DIRIGIDO**, en donde:
 - Cada **nodo** representa una o más sentencias secuenciales **y/o una ÚNICA CONDICIÓN** (así como los puntos de entrada y de salida de la unidad de programa)
 - * Cada **nodo** estará **etiquetado** con un **entero** cuyo valor será único
 - * Si un **nodo** contiene una **condición** anotaremos a su **derecha** dicha condición
 - Las **aristas** representan el flujo de ejecución entre dos conjuntos de sentencias (representadas en los nodos)
 - * Si un **nodo** contiene una **condición** etiquetaremos las aristas que salen del nodo con "T" o "F" dependiendo de si el valor de la condición que representa es cierto o falso.

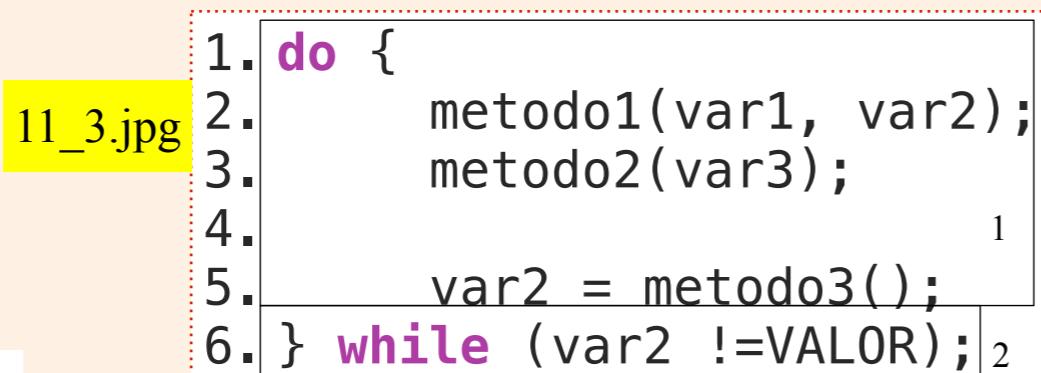
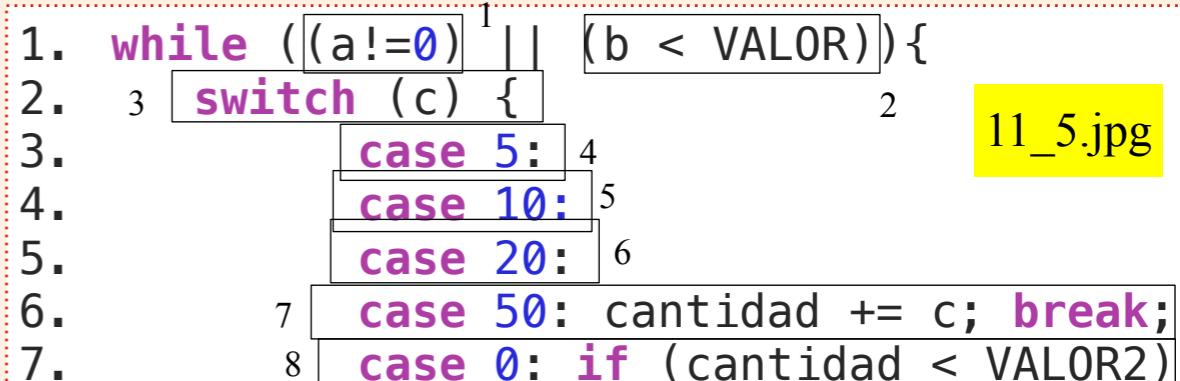
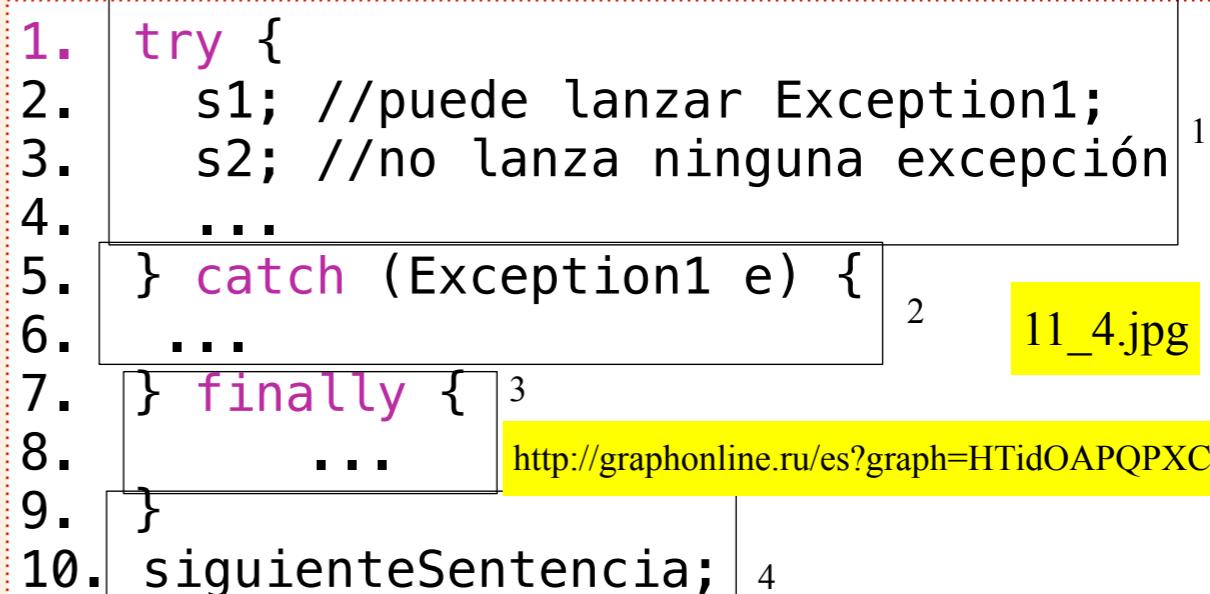
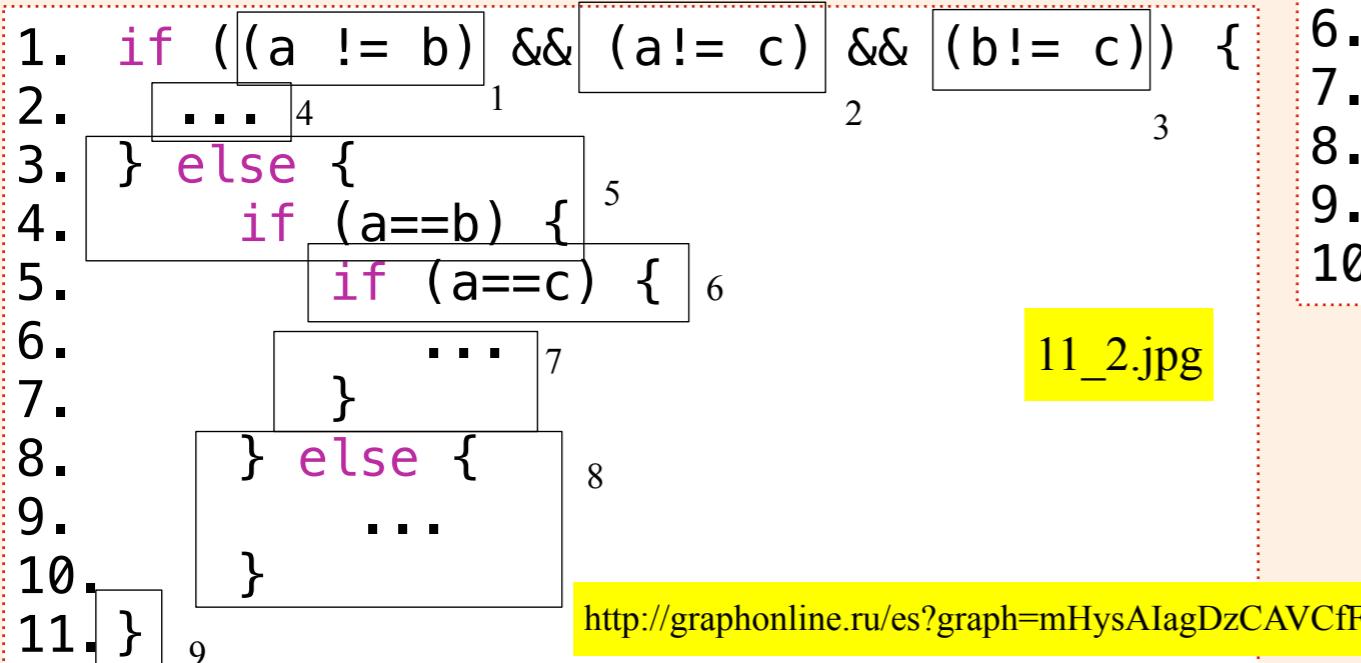
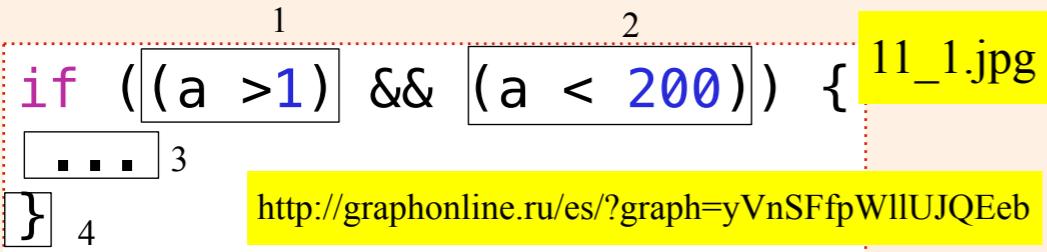
```
If(a > 20){  
    k = "valor correcto"  
} else {  
    k = "repita entrada"  
}
```



CONSTRUCCIÓN DE UN CFG (I)

S Hay que tener claro cómo funcionan las sentencias de control del lenguaje!!!

- Representa los grafos de flujo asociados a los siguientes códigos java:



Inténtalo!

CRITERIOS DE SELECCIÓN DE CAMINOS

Las pruebas EXHAUSTIVAS son IMPOSIBLES!!!

- **Estructuralmente** un camino es una secuencia de instrucciones en una unidad de programa (desde el punto de entrada, hasta el punto de salida)
- **Semánticamente** un camino es una instancia de una ejecución de una unidad de programa (comportamiento programado)
- Es necesario seleccionar un conjunto de caminos con algún criterio de selección. Algunos ejemplos son:
 - Elegimos todos los caminos del grafo
 - Elegimos el conjunto mínimo de caminos para conseguir ejecutar TODAS las SENTENCIAS al menos una vez
 - Elegimos el conjunto mínimo de caminos para conseguir ejecutar TODAS las CONDICIONES al menos una vez
- No generaremos entradas para los tests en las que se ejecute el mismo camino varias veces. Aunque, si cada ejecución del camino actualiza el estado del sistema, entonces múltiples ejecuciones del mismo camino pueden no ser idénticas

Cada método de diseño usa un criterio de selección diferente!!!!

Ese criterio depende de un OBJETIVO. Pero cualquier método de diseño proporciona un conjunto de casos de prueba efectivos y eficientes para ese objetivo!!!



MCCABE'S BASIS PATH METHOD

(Método del camino básico)

- Es un **método** de **DISEÑO** de pruebas de caja blanca que permite ejercitar (ejecutar) cada **camino independiente** en el programa
 - Fue propuesto inicialmente por Tom McCabe en 1976. Este método permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedural y usar esta medida como guía para la definición de un conjunto básico de caminos de ejecución
 - El método también se conoce como "Método del camino básico"

El **objetivo** del método es éste!!!



- Si ejecutamos **TODOS** los caminos independientes:

- Estaremos ejecutando **TODAS** las **sentencias** del **programa**, al menos una vez
 - Además estaremos garantizando que **TODAS** las **condiciones** se ejecutan **en sus vertientes** verdadero/falso

- ¿Qué es un **camino independiente**?

- Es un **camino** en un grafo de flujo (CFG) que difiere de otros caminos en al menos un nuevo conjunto de sentencias y/o una nueva condición (Pressman, 2001)

El **número** de **caminos independientes** determinará el **número** de **filas** de la **tabla**. Cada **fila** detectará **defectos** en un **determinado** **subconjunto** de **sentencias** del **programa** (ejercitando un **determinado** **comportamiento**)

DESCRIPCIÓN DEL MÉTODO

S Recuerda que debes ser sistemático a la hora de aplicar los pasos y tener claro para qué estamos haciendo cada uno de ellos!!!

- P
- P
1. Construir el grafo de flujo del programa (CFG) a partir del código a probar
 2. Calcular la complejidad ciclomática (CC) del grafo de flujo
 3. Obtener los caminos independientes del grafo
 4. Determinar los datos concretos de entrada (y salida esperada) de la unidad a probar, de forma que se ejercent todos los caminos independientes. El resultado esperado siempre se obtendrá en función de la ESPECIFICACIÓN de la unidad a probar

CONJUNTO PROGRAMADO

Tabla resultante del diseño de las pruebas:

Camino	Entrada 1	Entrada 2	...	Entrada n	Resultado Esperado
C1	d ₁₁	d ₁₂	...	d _{1n}	r ₁
...					
C2	d ₂₁	d ₂₂	...	d _{2n}	r ₂

Recuerda que los valores de entrada y salida deben ser CONCRETOS!!!

Una columna para CADA dato de entrada

Una columna para CADA dato de salida

COMPLEJIDAD CICLOMÁTICA

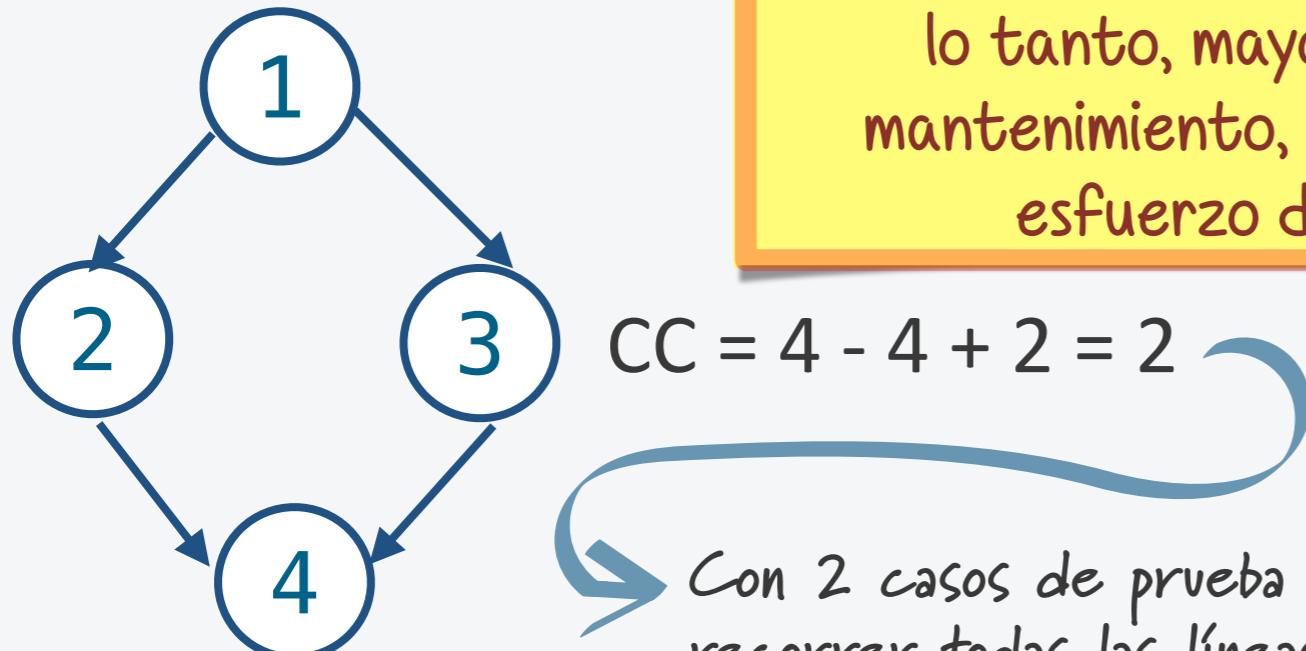
P

S

Determina el número de FILAS de la tabla!!

P

- Es una MÉTRICA que proporciona una medida de la complejidad lógica de un componente software
- Se calcula a partir del grafo de flujo:
$$CC = \text{número de arcos} - \text{número de nodos} + 2$$
- El valor de CC indica el MÁXIMO número de caminos independientes en el grafo
- Ejemplo:



$$CC = 4 - 4 + 2 = 2$$

Con 2 casos de prueba podemos recorrer todas las líneas y todas las condiciones en sus vertientes verdadera y falsa

A mayor CC, mayor complejidad lógica, por lo tanto, mayor esfuerzo de mantenimiento, y también mayor esfuerzo de pruebas!!!

El valor máximo de CC comúnmente aceptado como "tolerable" es 10

Se puede reducir la complejidad lógica refactorizando el código para incrementar el nivel de abstracción (modularizar)

OTRAS FORMAS DE CALCULAR CC

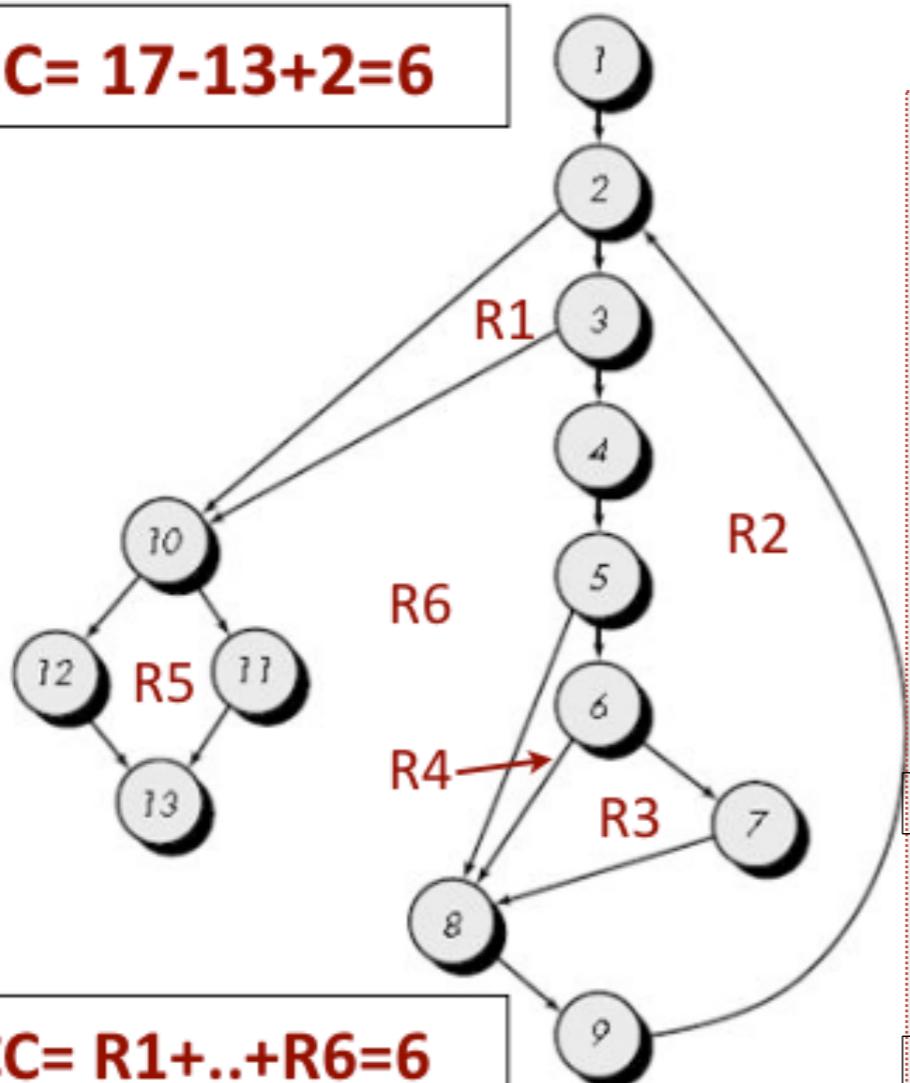
CC = número de arcos – número de nodos + 2

CC = número de regiones

CC = número de condiciones + 1

¡CUIDADO!: Las dos últimas formas de cálculo son aplicables SOLO si el código es totalmente estructurado (no saltos incondicionales)

$$CC = 17 - 13 + 2 = 6$$



$$CC = R1 + \dots + R6 = 6$$

```

...
i=1;
total.input=total.valid=0; 1
sum=0;
while ((value[i] <> -999) && (total.input<100)) { 3
    total.input+=1; 2
    if ((value[i]>= minimum) && (value[i]<= maximum)) { 6
        total.valid+=1; 5
        sum= sum + value[i]; 7
    }
    i+=1; 8
}
if (total.valid >0) { 10
    average= sum/total.valid; 11
} else average = -999; 12
return average; 13

```

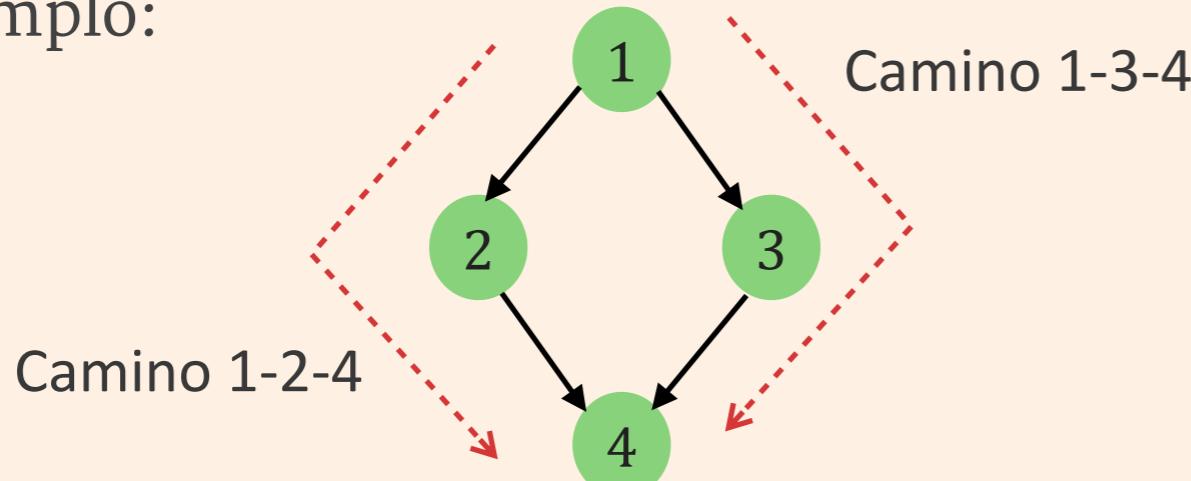
$$CC = 5 + 1 = 6$$

CAMINOS INDEPENDIENTES

Cada camino independiente recorre un nodo o una arista (como mínimo) que no se había recorrido antes!!!

- Buscamos (como máximo) tantos caminos independientes como valor obtenido de CC
 - Cada camino independiente contiene un nodo, o bien una arista, que no aparece en los caminos independientes anteriores
 - Con ellos recorreremos TODOS los nodos y TODAS las aristas del grafo

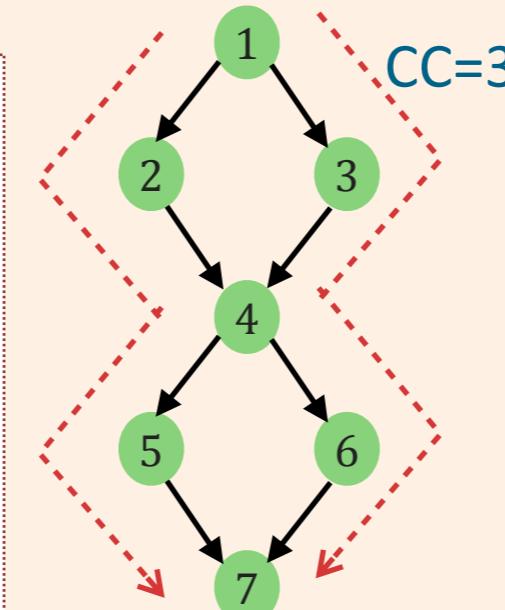
- Ejemplo:



- Es posible que con un número inferior a CC recorramos todos los nodos y todas las aristas

- Ejemplo:

```
if (a>=20) {  
    result=0;  
} else {  
    result=10;  
}  
  
if (b>=20) {  
    result=0;  
} else {  
    result=10;  
}
```



opción 1:

C1 = 1-3-4-6-7
C2 = 1-3-4-5-7
C3 = 1-2-4-6-7

opción 2:

C1 = 1-3-4-6-7
C2 = 1-2-4-5-7

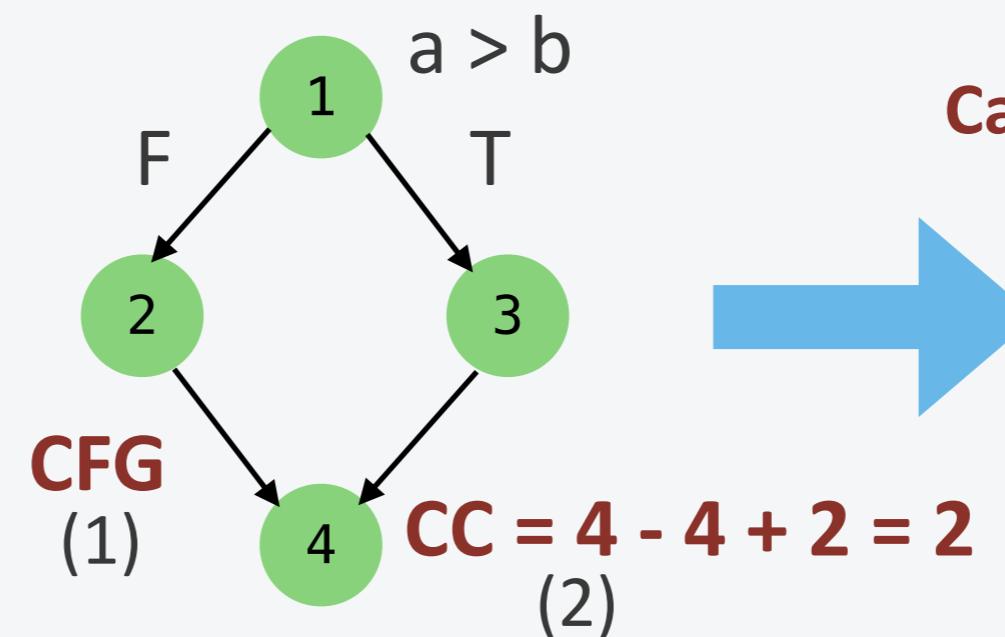
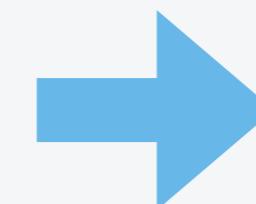
Ambas opciones son válidas

EJEMPLO DE APLICACIÓN DEL MÉTODO

Es muy importante ser sistemático pero sabiendo lo que haces en cada paso!!!

Método que compara dos enteros a y b, y devuelve 20 en caso de que el valor a sea mayor que b, y cero en caso contrario:

```
if (a > b) {  
    result = 20;  
} else {  
    result = 0;  
}
```



(3)

Caminos independientes

$$C1 = 1-3-4$$
$$C2 = 1-2-4$$

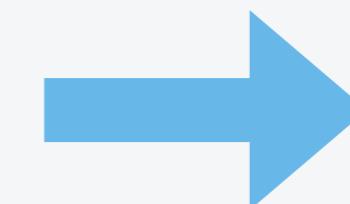


Tabla resultante del diseño de casos de prueba



Camino	Datos Entrada		Resultado Esperado
C1	a = 20	b = 10	result = 20
C2	a = 10	b = 20	result = 0

(4) Valores de entrada

Resultado esperado

(5)

SIEMPRE son valores CONCRETOS!!!

EJEMPLO: BÚSQUEDA BINARIA

P

S

P

```
//Asumimos que la lista de elementos está ordenada de forma ascendente
class BinSearch
public static void search (int key, int [ ] elemArray, Result r) {
    int bottom = 0; int top = elemArray.length -1; NODO 0
    int mid; r.found= false; r.index= -1;
    while (bottom <= top) { INICIO WHILE OBLIGADO NODO 1
        mid = (top+bottom)/2;
        if (elemArray [mid] == key) { NODO 2
            r.index = mid;
            r.found = true;
            return;
        } else { NODO 4
            if (elemArray [mid] < key)
                bottom = mid + 1; NODO 5
            else top = mid -1;
        }
    } //while loop NODO 7 - FIN WHILE Y SALIDA IF
} //search FIN NODO 8
//class
```

WHITE
OJO!!! DEL 7 AL 8
NO HAY ARCO

<http://graphonline.ru/es?graph=VoEovnnpPRuxnkFR>

(Es lo mismo que la diapositiva 21)

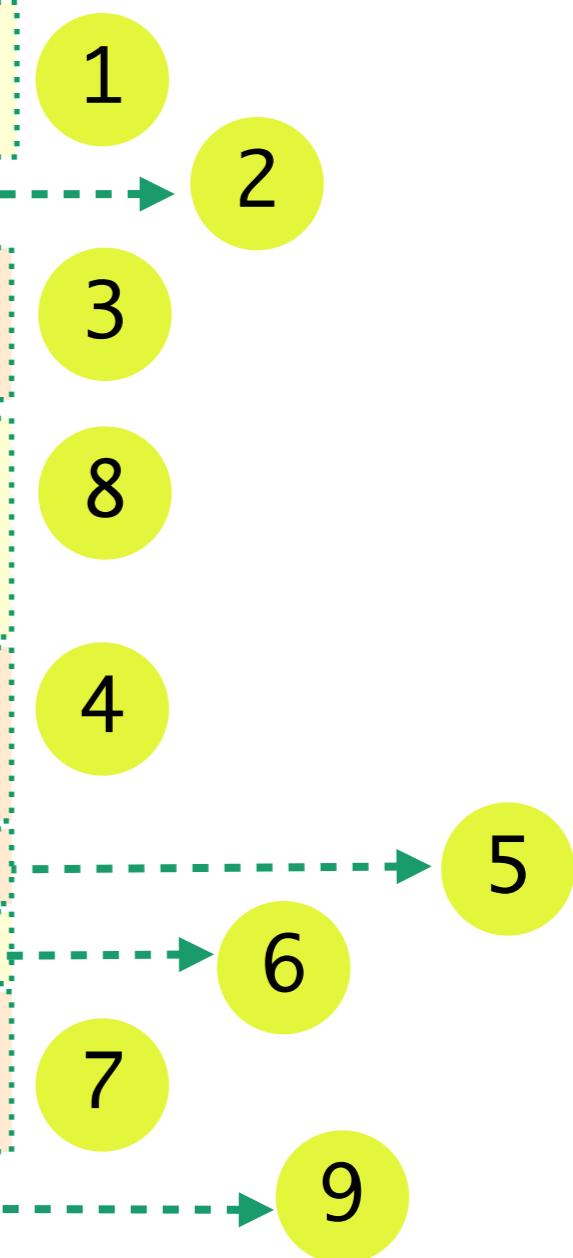
Especificación del método search():
Dado un vector de enteros ordenados ascendentemente, y dado un entero (key) como entrada, el método search() busca la posición de key en el vector y devuelve el valor found=true si lo encuentra, así como su posición en el vector (dada por index). Si el valor de key no está en el vector, entonces devuelve el valor found=false

VAMOS A IDENTIFICAR LOS NODOS DEL GRAFO

//Asumimos que la lista de elementos está ordenada de forma ascendente

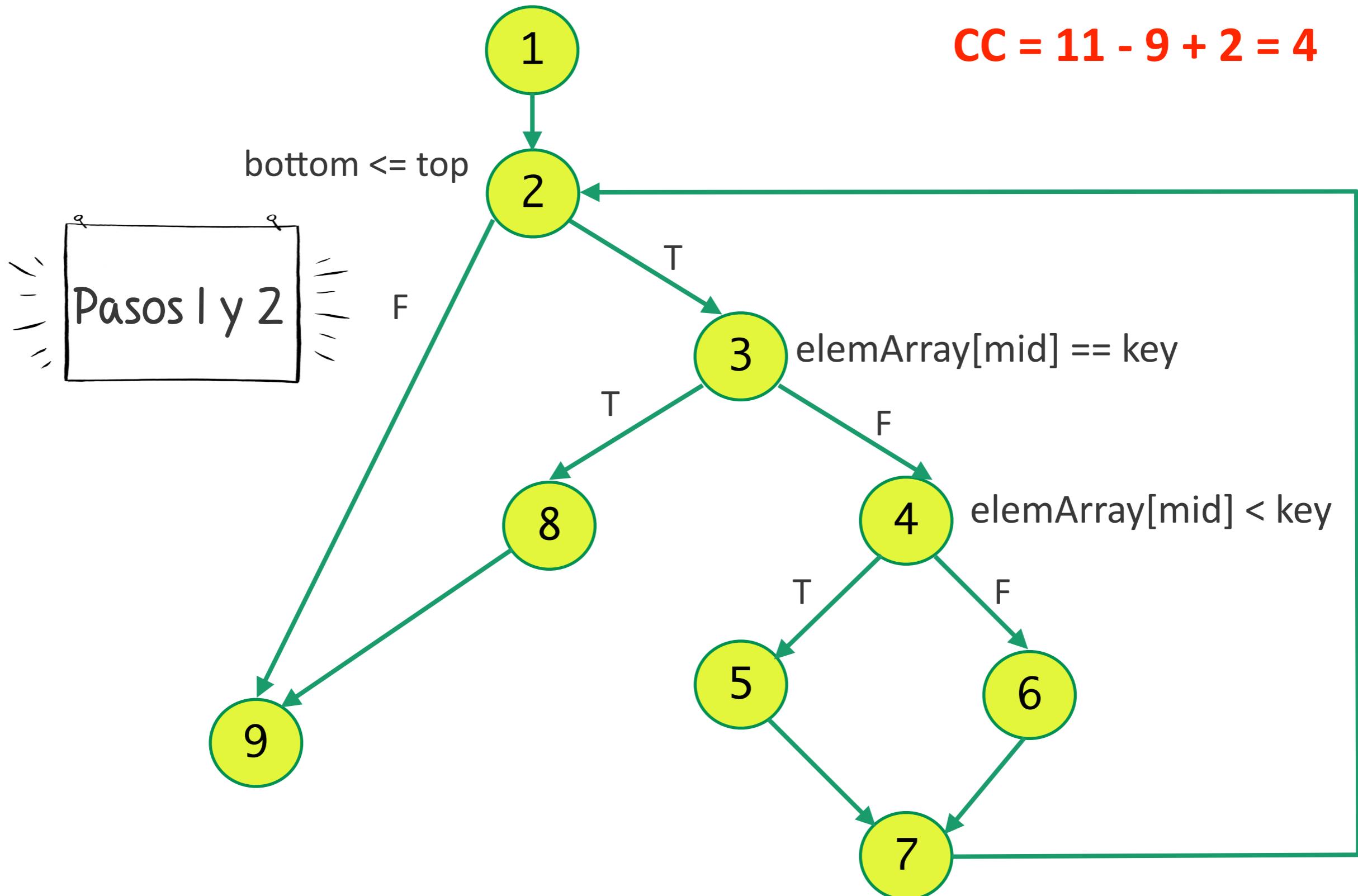
class BinSearch

```
public static void search (int key, int [ ] elemArray, Result r)
{   int bottom = 0;      int top = elemArray.length -1;
    int mid;           r.found= false; r.index= -1;
    while (bottom <= top) {
        mid = (top+bottom)/2;
        if (elemArray [mid] == key) {
            r.index = mid;
            r.found = true;
            return;
        } else {
            if (elemArray [mid] < key)
                bottom = mid + 1;
            else top = mid -1;
        }
    } //while loop
} //search
} //class
```



GRAFO ASOCIADO Y VALOR DE CC

$$CC = 11 - 9 + 2 = 4$$



CAMINOS INDEPENDIENTES

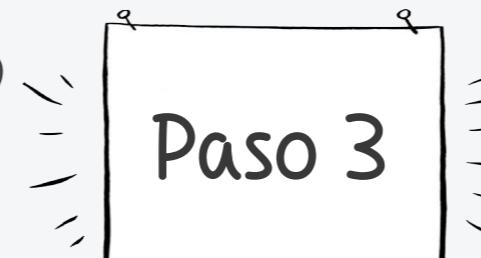
P

S

- Posible conjunto de caminos independientes

P

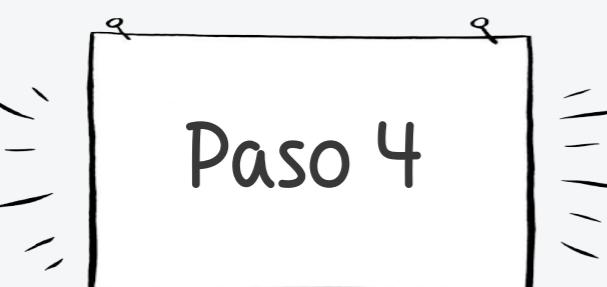
- C1: 1, 2, 3, 4, 6, 7, 2, 9
- C2: 1, 2, 3, 4, 5, 7, 2, 9
- C3: 1, 2, 3, 8, 9



En este ejemplo, con tres caminos podemos recorrer todos los nodos y todas las aristas

- Ejercicio: Calcula la tabla resultante:

Camino	Datos Entrada		Resultado Esperado	
	key	elemArray	r.found	r.index
C1	2	[3,4]	false	?
C2	5	[1,4]	false	?
C3	2	[1,2,3]	true	1



Para indicar el valor del resultado esperado necesitamos conocer la ESPECIFICACIÓN del método



EJERCICIOS PROPUESTOS (I)

P

S

P

○ Calcula la CC para cada uno de estos códigos Java:

```
public int divide(int numberToDivide, int numberToDivideBy) throws BadNumberException{
    if(numberToDivideBy == 0){ 1
        throw new BadNumberException("Cannot divide by 0");
    }
    return numberToDivide / numberToDivideBy; 3
} 4
```

<http://graphonline.ru/es?graph=sRrLZOrNwMeCZTUI>

$$\begin{aligned} \text{CC} &= \text{arcos} - \text{nodos} + 2 \\ \text{CC} &= 4 - 4 + 2 = 2 \end{aligned}$$

Código 1

23_1.jpg

```
public void callDivide(int m,int n){
    try {
        int result = divide(m,n); 1
        System.out.println(result);
    } catch (BadNumberException e) { 2
        //do something clever with the exception
        System.out.println(e.getMessage());
    }
    System.out.println("Division attempt done"); 3
} 4
```

<http://graphonline.ru/es?graph=UDTXfgIIIyUyZjQv>

CC = 4 - 4 + 2 = 2

Código 2

23_2.jpg

```
public void openFile(){
    try {
        // constructor may throw FileNotFoundException
        FileReader reader = new FileReader("someFile");
        int i=0; 1
        while(i != -1){ 2
            //reader.read() may throw IOException
            i = reader.read();
            System.out.println((char) i );
        } 3
        reader.close();
        System.out.println("--- File End ---"); 5
    } catch (FileNotFoundException e) { 6
        //do something clever with the exception
    } catch (IOException e) { 7
        //do something clever with the exception
    }
} 8
```

Código 3

23_3.jpg

OJO!! la declaración de i NO puede ir junto al nodo del TRY porque es una sentencia que no lanza una excepción

Se puede hacer en 3, por eso sale la CC = 4

CC = 11 - 8 + 2 = 4

EJERCICIOS PROPUESTOS (II)

S

P

Diseña los casos de prueba para el método validar_PIN(), cuyo código es el siguiente:

P

```
1. public class Cajero {  
2.     ...  
3.     public boolean validar_PIN (Pin pinNumber) {  
4.         boolean pin_valido= false;  
5.         String codigo_resuesta="GOOD";  
6.         int contador_pin= 0;  
7.         ...  
8.         while (( !pin_valido) 2&& (contador_pin <= 32) &&  
9.                 !codigo_resuesta.equals("CANCEL")) {  
10.             codigo_resuesta = obtener_pin(11. pinNumber);  
11.             if (!codigo_resuesta.equals("CANCEL")) {  
12.                 pin_valido = comprobar_pin(pinNumber);  
13.                 if (!pin_valido) {  
14.                     System.out.println("PIN inválido, repita");  
15.                     contador_pin=contador_pin+1;  
16.                 }  
17.             }  
18.         } 8  
19.         return pin_valido; 9  
20.     } NODO FIN → 10  
21.     ...  
22. }
```

http://graphonline.ru/es?graph=DWzVgiiDOYvgpruF

CC = 14 - 10 + 2 = 6

la especificación la tenéis a continuación



EJERCICIOS PROPUESTOS (II) (CONTINUACIÓN)

PIN.jpg

Especificación del método validar_PIN():

- P
- El método validar_PIN() anterior valida un código numérico de cuatro cifras (objeto de la clase Pin). Dicho código se obtendrá después de introducirlo a través de un teclado (asumimos que en el teclado solamente hay teclas numéricas (0..9), y una tecla para cancelar). Si el usuario **pulsa** en algún momento la tecla de **cancelar**, entonces la **validación** se considerará "**false**". El usuario dispone de **tres intentos** para introducir un pin válido, en cuyo caso el método validar_PIN() devuelve cierto, así como el número de pin, y en **caso contrario** devuelve **falso**.

Nota: La introducción del código numérico se ha implementado en otra unidad (el método obtener_pin()), que se encargará de “leer” el código introducido por teclado creando una nueva instancia de un objeto Pin, y devuelve “**GOOD**” si **no** se pulsa la **tecla para cancelar**, o “**CANCEL**” si se ha **pulsado** la **tecla para cancelar** (carácter ‘\’).

Además usamos el método comprobar_pin(), que **verifica** que el código introducido tiene **cuatro cifras** y se corresponde con la **contraseña** almacenada en el sistema para dicho usuario, devolviendo **cierto** o **falso**, en función de ello.

Recuerda que el **método del camino básico** sólo lo aplicaremos a nivel de **UNIDAD!!**



Hemos definido una unidad como
UNIDAD = MÉTODO JAVA

Y AHORA VAMOS AL LABORATORIO...

P

P El proceso de diseño lo haremos "manualmente"

Diseñaremos casos de prueba utilizando el método del CAMINO BÁSICO

Hay que tener claros TODOS los pasos!!!

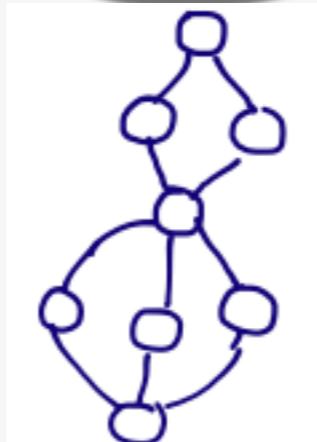
```
package ppss;

public class Matricula {
    public float calculaTasaMatricula(int edad,
        boolean familiaNumerosa,
        boolean repetidor) {
        float tasa = 500.00f;

        if ((edad <= 25) && (!familiaNumerosa) || (!repetidor)) {
            tasa = tasa + 1500.00f;
        } else {
            if ((familiaNumerosa) || (edad > 65)) {
                tasa = tasa / 2;
            }
            if ((edad >= 50) && (edad < 65)) {
                tasa = tasa - 100.00f;
            }
        }
        return tasa;
    }
}
```

unidad a probar

CFG



CC

CC = ...

tabla de casos de prueba

caminos independientes

C1: 1-2-4-... -14

C2: 1-3-6-... -14

...

CN: 1-2-7-... -14

(N ≤ CC)

Camino	DATOS DE ENTRADA							
C1	d ₁₁	d ₁₂	...	d _{1q}	r ₁₁	...	r _{1k}	
...								
CN	d _{n1}	d _{n2}	...	d _{nq}	r _{n1}	...	r _{nk}	

ESTA TABLA La utilizaremos en la siguiente práctica!!!...

REFERENCIAS BIBLIOGRÁFICAS



- A practitioner's guide to software test design. Lee Copeland. Artech House Publishers. 2004
 - Capítulo 10: Control Flow Testing
- Pragmatic software testing. Rex Black. Wiley. 2007
 - Capítulo 21: Control Flow Testing
- Software testing and quality assurance. Kshirasagar Naik & Priyadarshi Tripathy. Wiley. 2008
 - Capítulo 4: Control Flow Testing